High Performance Computing - HW 5

Anthony Maylath

April 27, 2019

The Courant Institute for Mathematical Sciences, New York University

1 Question 1

My implementation contains the function *timeRing*() to capture the latency of message passing over integers and *timeRingAr*() to capture the bandwidth over an integer array of size 500,000. Each implmentation adds the rank of the sending process for each send executed in the function. For *timeRingAr*() the rank is added to every element in the array. Hence, the time to compute these increments are included in the latency and bandwidth figures.

To test the performance of my code, I ran it with a varying number of processes on crunchy5 and crunchy6. I ran the code with the following command:

mpirun -np [num processes] –oversubscribe -H crunchy5.cims.nyu.edu, crunchy6.cims.nyu.edu ./int_ring -i [num iterations]

Table 1 shows the performance of my code over various number of processes and iterations. I ran my code on CIMS between 2 and 2:30pm on Tuesday. To better understand the results, the reader can cross check load on these machines during these times. Time seems to go up when the number of processes or number of iterations goes up. This makes sense as the number of processes increases the ring size. Latency and Bandwidth tend to increase with number of processes as messages need to communicate between more cores. Both Latency and Bandwidth tend to be smallest when number of iterations and number of processes are high. This occurs as the MPI overhead is small compared to the number of work in this instance.

Processes	Iterations	Time Primative	Time Array	Latency	Bandwidth (GB/s)
2	1	0.000278	0.036327	0.000139	0.041291
10	1	0.003824	0.062924	0.000382	0.023838
30	1	0.049308	0.248918	0.001644	0.006026
2	100	0.021569	4.608056	0.000108	0.000326
10	100	0.136884	7.396445	0.000137	0.000203
30	100	0.696036	53.031526	0.000232	0.000028
2	1000	0.318983	38.538305	0.000159	0.000039
10	1000	0.576416	77.588472	0.000058	0.000019

Table 1: Performance on crunchy5 and crunchy6

2 Question 2

Week	Work	Who
4/8-4/14	Identify Datasets for clustering algorithm	Anthony Maylath and John Donaghy
	Build datareader to read dataset	John
4/15-4/21	Code Serial k-means and verify with sklearn python script	Anthony
	Implement Initial MPI Parallel k-means	John
	Debug Initial MPI implementation to match serial version.	John and Anthony
	Optimize algorithm to use fewer message passes	John and Anthony
	Implement AVX logic to compute new clusters	John
4/22-4/28	Upgrade data parser to handle larger datasets	John
	Set up scratch disk	John
	Research MPI implementations for EM algorithm	Anthony
4/29-5/5	Run scalability checks on big data. Check k-means performance	Anthony and John
	(Time permitting) Implement serial EM algorithm. Verify with python	Anthony
	Final optimizations on k-means MPI implementation	John and Anthony
5/6-5/12	Start drafting report and slides	John and Anthony
	Build MPI version of EM algorithm (time permitting)	John and Anthony
	Go to office hours to get advise on how to improve things	John and/or Anthony
	Finalize k-means implemenation	John and Anthony
5/13-5/19	(Time permitting) Finalize EM algorithm and compare to k-means	John and Anthony
	Run final tests on code base	John and Anthony
	Finish report and slides	John and Anthony

Table 2: MPI Clustering Algorithms